Review-2

Amer Diwan

Type inference

• Idea: compute types (semi-)automatically
  – programming language or non-standard
• Type inference in SML
  – Almost completely automatic
  – Polymorphic
• Type inference in Steensgaard
  – Automatic
  – Monomorphic
Type inference (cont.)

- Type inference in Lackwit
  - Automatic
  - Polymorphic
- Other keywords: flow insensitive

Types to optimize programs

- Idea: type safety can be used to improve program performance
- Types for resolving method invocations
  - Types yield a pretty good analysis
  - Incorporating some instruction analysis helps
- Types for pointer analysis
  - Types alone yield a pretty effective analysis
  - Incorporating instruction analysis does not help
- Other keywords: Redundant load elimination
First class functions

• Idea: can treat functions as regular old values
• How incorporated in languages
  – Weakly (e.g., C, C++)
  – A little better (e.g., Modula-3)
  – Full glory (SML)
• Implications: fully supporting first-class functions can force one to put activation records on heap
• Other keywords: nested functions, closures, static link, environment, escaping functions

Exception handling

• Idea: a “non-local” jump from detection of error to code that knows how to handle the error
• How implemented in languages
  – Termination, statically scoped (Java, Modula-3, C++)
  – Resumption, dynamically scoped (PL/1)
• Implication: direct costs may be low but indirect costs may be high
• Other keywords: exception table, throw, raise, catch
Continuations

- **Idea:** a continuation captures the “rest of the computation”. Throwing a continuation restores some previous state.
- **How implemented in languages**
  - Strangely (C’s setjmp/longjmp)
  - callcc (SML, some LISP dialects)
- **Implications:** may cause activation records to be allocated on the heap. Can be smart about it, though!
- **Other keywords:** throw, lazy copying

Continuation example 1

- fun product l = callcc(fn exit =>
  let fun prod_aux nil = 1
  | prod_aux(0::t) = throw exit 0
  | prod_aux(h::t) = h * prod_aux t
  in prod_aux l
  end)
- **exit:** a continuation that captures what happens to the result of product
Continuations for co-routines

• Assume that there are two arrays, A and B
  – each element has three parts: an integer value, a “put” semaphore, and a “get” semaphore
  – thread 1 reads from array A, doubles the value and writes it in array B
  – thread 2 reads from array B, halves the value and puts it in array A
  – both use the get and put semaphores for exclusive access

What do P and V do?

• Let g be a get semaphore and p be a put semaphore associated with element A[i]
• P g: wait until a new value arrives in A[i]. Lock A[i].
• V g: signal that the value in A[i] has been read
• P p: wait until allowed to write a new value to A[i]. Lock A[i]
• V p: signal that a new value may be written in A[i]
Code for thread 1

- fun thread1(i,j) =
  let val {value=Av, put=Ap, get=Ag} = A sub i
  val {value=Bv, put=Bp, get=Bg} = B sub j
  val x = (P Ag; !Av)
  in V Ap; P Bp; Bv = 2*x; V Bg;
  yield();
  thread1((i+1) mod Alen, (j+1) mod Blen)
end

The magic is in the “yield”

- fun yield() =
  if random()
  then ()
  else callcc(fn k => (enqueue k; dispatch()))
- fun dispatch() = let val head::rest = !queue
  in queue := rest; throw head()
What happens when a continuation is captured?

What happens when underflow' executes?

Underflow' executes
What happens when a continuation is thrown?

Copy the top segment of the continuation, and set the PC to RA2